Asynchronous Programming in Rust

An Empirical Analysis on the Usage of Asynchronous Code by Rust Developers

Benjamin Dockx

Supervisors: Andreea Costea[1], Ruben Backx[1]

[1]**EEMCS, Delft University of Technology, The Netherlands**

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 22, 2026

Name of the student: Benjamin Dockx
Final project course: CSE3000 Research Project
Thesis committee: Andreea Costea, Ruben Backx, Przemyslaw Pawelczak

Abstract

Asynchronous programming in Rust introduces challenges regarding correctness, synchronization and concurrency. Existing literature has focused on uncovering bugs and providing verification techniques to mitigate these. However, further research within this field is still required and to the best of our knowledge, limited empirical data exists analyzing why asynchronous Rust is used in practice and the prominence of different runtimes.

In this paper, we present Rustc-Analysis, an automated analysis tool for discovering and extracting underlying structural information of asynchronous Rust code. We subsequently use this tool to analyze runtime adoption and characterize the functionality provided by asynchronous APIs.

Our analysis shows that asynchronous APIs are mainly used for I/O-related tasks, particularly networking and file-system operations, as well as synchronization between concurrent tasks. Furthermore, we find tokio to be the most widely adopted runtime, followed by futures, async-std and smol.

1 Introduction

Introduction to Rust. Rust is a statically typed programming language first released in 2015. Through its ownership and borrowing model, Rust provides safety guarantees like those often associated with high-level languages, while maintaining performance comparable to low-level languages like C and C++ [9]. These properties have contributed to Rust's growing popularity among software developers [22]. As Rust has matured, asynchronous programming has become an important element within its ecosystem, enabling applications to manage large numbers of concurrent operations.

Research Gap. Asynchronous Rust code allows applications to efficiently handle blocking operations and perform tasks in parallel. However, it also adds a time dimension which developers must take into account when writing their code. Schoeder et al. [21] show asynchronous code and tests are one of the major contributors for flakiness in GitHub pipelines, indicating Rust async still challenges developers.

Research regarding concurrent Rust code has mainly focused on safety and verification techniques. Qin et al. [17; 18] performed an empirical analysis of open-source Rust code, looking at concurrency bugs, providing static bug detectors and demonstrating these by discovering new bugs. Pearce et al. [15] propose RustMC, a stateless model checker for discovering concurrency bugs in unsafe Rust code.

Performance-related investigations are comparatively limited and have often been community driven [11; 23]. While these provide insight into runtime behavior and implementation overheads, little empirical knowledge exists on how async Rust is used in practice. We have a limited understanding regarding the popularity of runtimes and the functionality provided through asynchronous APIs. We aim to address these questions in this research paper.

Research Objectives. This study aims to provide an empirical analysis of asynchronous programs in the Rust ecosystem, looking specifically at runtime and API usages. In doing so, we present Rustc-Analysis, a tool aimed at analyzing concurrent Rust code, and use it to answer our research questions:

RQ1: What are the most popular runtimes?

RQ2: What functionality do external async APIs provide?

Approach Overview. We fetched open-source Rust projects from GitHub and analyzed their direct and indirect runtime usages. We then used Rustc-Analysis in combination with the Rust compiler to create call-graph representations of their source code. Using this, we identified the asynchronous API usages and subsequently classified them according to the functionality they provided.

Artifact Availability. The source code, dataset and analysis scripts used in this study, alongside the necessary steps to reproduce our findings, are publicly available in our recreation package: https://GitHub.com/Benjamin1260/rustc-analysis.

2 Background

2.1 Executors & Runtimes

Rust is unique in that it does not provide an inherent implementation for executing its own `Futures`. This functionality is provided through an executor. At its core, an executor simply polls `Futures` repeatedly, until they complete. We explain this further in section 2.2. Developers may write their own executor or use existing libraries which provide this functionality, the most prominent of which are tokio, futures, async-std and smol. More libraries exist which are able to run `Futures`, but these are more niche and not commonly used.

2.2 Async Await Semantics

Rust allows users to write their own concurrent code using the `async` and `await` keywords. This is syntactic sugaring which create and use the `Future` trait under the hood [28]. An example of the desugaring of `async` is shown in listing 1.

```rust
// Source
async fn example(x: &str) -> usize {
    x.len()
}
```

```rust
// Desugared
fn example<'a>(x: &'a str) ->
    impl Future<Output = usize> + 'a
{
    async move { x.len() }
}
```

Listing 1: Desugaring of the `async` keyword [28]

These `Futures` [34] can be seen as state machines representing a computation block, waiting to be run. This is different from other languages (like `JavaScript`), in that they do not automatically start running upon creation. For this reason, they can be seen as "lazy" [13].

Now knowing how `Futures` are created, it still leaves the question: how do we `await` them [29]? To explain this, a rough expansion of the `await` keyword is shown in listing 2.

```
let mut future =
    IntoFuture::into_future($expression);
let mut pin = unsafe {
    Pin::new_unchecked(&mut future)
};
loop {
    match Future::poll(
        Pin::borrow(&mut pin), &mut ctx
    ) {
          Poll::Ready(item) => break item,
          Poll::Pending     => yield,
    }
}
```

Listing 2: Expansion of `await` as per the original RFC [24]

When a `Future` [35] is awaited, we start repeatedly polling it. This makes it begin or continue running from its current state. If it contains an internal `.await`, the outer `Future` polls the inner `Future`. If the inner `Future` returns `Poll::Ready`, the outer `Future` continues its computations. If the inner `Future` returns `Poll::Pending`, the outer `Future` propagates this and also returns `Poll::Pending`.

The `poll` contains the outer `Future`'s `context` with its `waker` function [25]. This `waker` is called when progress can be made again. It schedules the outer `Future` to be polled again by the executor, upon which it will continue running.

Alternatively, the `Future` may be able to continue running until a return statement. This means it completed its computation, upon which it returns `Poll::Ready` and finishes.

2.3 Syntax vs. Semantics

Since a large part of our analysis will be automated, it becomes important to acknowledge the capabilities of an automated approach. This is mainly limited through its syntactical vs. semantical understanding of the code.

Syntax [38] involves a high-level knowledge of what the text looks like without actually understanding its meaning. To demonstrate this, consider: “a warm tree sleeps vigorously”. A syntactical approach would not see anything wrong with this, as it would simply look at a set of grammar rules and conclude that it is grammatically sound.

A semantical [37] approach implies a deeper understanding of the actual meaning contained within the text. In the context of our source code, it would mean actually understanding what the program is doing, distinguishing individual expressions, detecting function calls, type resolution and more.

Given this brief explanation, we hope it becomes clear why the semantical approach may be considered more powerful and preferred for performing a more advanced analysis.

2.4 Rust Representations

For our static analysis, it is relevant to understand the different ways in which Rust programs may be represented [30]. An overview of this can be seen in figure 1.

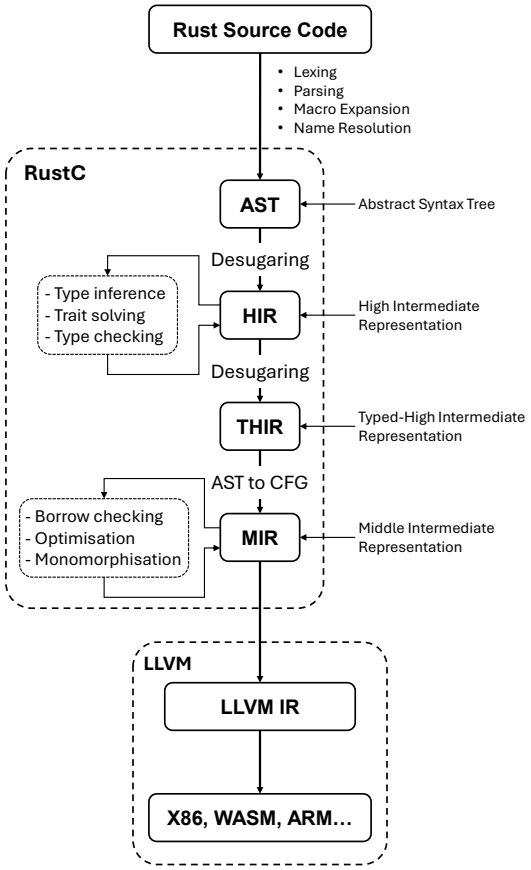


Figure 1: RustC Representation Layers. Adapted from [19].

The most accessible representations are the source code, Abstract Syntax Tree (AST), High Intermediate Representation (HIR) and Middle Intermediate Representation (MIR).

Starting off with the **source code**, this is simply the raw input to the Rust compiler in the shape of a `.rs` file written by the programmer. This is the shape of our input data.

This gets turned into the **AST** [32], which is simply a structured version of the source code. It does not encapsulate any semantics and is simply a more machine-friendly representation that the compiler uses.

The AST is lowered into the **HIR** [33], allowing more advanced semantic analysis. This representation still strongly resembles the input source code, while already including some desugaring. This includes `async` functions, which are turned into regular functions returning opaque types implementing `Future`. In our analysis specifically, HIR nodes are used to access and iterate the input source code.

The final representation relevant to our research is the **MIR** [27]. It is very different from previous layers in the sense that it represents a Control Flow Graph (CFG) rather than an AST. In this CFG [36], all functions consist of basic blocks containing statements, with a terminator at the end allowing for (conditional) jumps. This representation contains the most semantic information and since type checking and method resolution have already been performed, function calls can be linked to their corresponding definition. This allows us to analyze function calls and create a call graph which plays an important part in our analysis later.

2.5 Internal vs. External

Within our research paper, we will often refer to functions, closures and crates as being internal or external. This refers to the location of their definition. Internal structures are defined within the source code we are currently analyzing, while external structures are provided by the repositories' dependencies. This distinction is relevant as it determines the structures being analyzed and our categorization of external APIs.

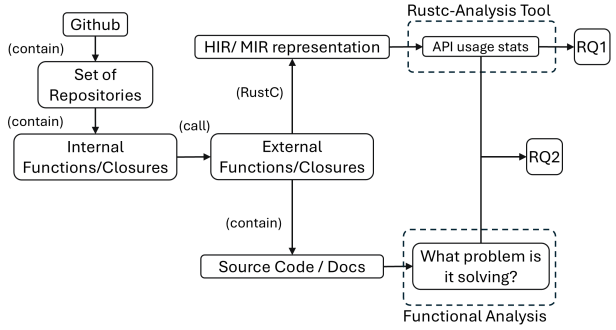


Figure 2: Overview of Data Flow in our Analysis

3 Methodology

We distinguish this section from the next in the following way: section 3 explains the approach we used to answer our research questions while section 4 explains the implementation details of the Rustc-Analysis tool we developed.

3.1 Overview of Analytical Process

Before explaining our data flow and how we tackled our research questions, we will share a brief high-level overview of the process. A visual representation of this process can be found in figure 2.

For our data source, we have chosen GitHub. From it, we get a collection of repositories, consisting of internal crates, containing Rust source code. This is the main input to our analysis, which is split up into two distinct parts: an automated structural analysis and a manual functional analysis. Each of these is responsible for filling in a different part of our analysis output file. This output file will take the shape of a relational database using DuckDB, visualized in figure 3.

Finally, we run SQL queries on this file to discover interesting insights and the answers to our research questions.

All of the repositories (link + commit hash), scripts and analysis tools we used as well as a step-by-step guide to reproduce our findings are available in our public recreation package hosted on GitHub [5].

3.2 Data Source

The goal of this study was to analyze how Rust developers use asynchronous Rust. We therefore decided to look specifically at open-source Rust projects aimed to serve regular end users. Furthermore, we decided to analyze popular projects, since we assume these to be important and to capture a large number of developers' inputs, whereas smaller projects might be developed by a single person. To obtain this dataset, we fetched GitHub Rust repositories ordered by their star count.

3.3 Runtime Analysis

To answer RQ1, we will analyze runtime usages in three distinct ways. First, we will analyze the internal usages of runtimes within our dataset. This is referring to a dependency where the source code of our dataset directly calls or uses functionality provided by one of these runtimes. Secondly, we include indirect dependencies, this implies runtime usages not only by our source code, but also by the dependencies of our source code. For example, our code might depend on reqwest, a high level HTTP client library, which in turn depends on a runtime like tokio. Finally, we analyze `crates.io`, which provides reverse-dependency-lookup functionality. This basically allows us to see and count the dependents of different crates, including runtimes.

3.4 Static Analysis

The bulk of our analysis is carried out by an automated tool we developed during the research project. It is responsible for providing the bulk of data we will later end up using in our SQL queries to answer our research questions.

We will use our tool to automatically fetch an initial list of repositories from GitHub. We then manually verify that each of these repositories actually compiles successfully. If any of these should fail, we will spend a maximum of 15 minutes trying to fix the failing build by for example resolving dependencies or adding necessary cargo arguments. If after 15 minutes we are still not able to compile the repository, we consider it broken and exclude it from our dataset.

Furthermore, we will limit our analysis strictly to Rust applications. Repositories like rust-lang/rust and awesomerust will be excluded since these are not applications.

After using this automated analysis tool, we will be left with a DuckDB file containing extensive information about the source code. It will contain information about the internal and external crates used, the internal and external functions and how these relate to one another through invocations.

3.5 Functional Analysis

To answer RQ2, our DuckDB file still needs categorizations. To be more precise, the functions returned by our analysis still should be annotated according to the problem they solve.

This is a categorization problem and creating a rule-based algorithm to solve this automatically would be infeasible. We considered using machine learning for this, which seemed like a justifiable and scalable strategy to our problem. Unfortunately, due to time constraints, this became infeasible and thus, we have decided to include it in subsection 8.4 as a suggestion for future work instead.

These external functions will be categorized in the following fashion: we start from an empty categorization tree. Each node in this tree will have a ruleset defining its children's categorizations. While analyzing functions, we first traverse the existing nodes according to their rules. If the function does not fit any existing node, a new ruleset is created and a new node is inserted into the tree. Furthermore, if an existing node contains a lot of functions and can be made more specific, we insert new child nodes with more specific rulesets. Whenever a new node is added to a parent node, all functions assigned to the parent node must be reevaluated to consider if they fit the new child node's criteria better. We will not enforce all functions to be assigned to a leaf node, as sometimes it makes sense to add specific child categorizations, while keeping some functions that do not fit any of the more specific children, in the parent node.

The rulesets we define will refer mainly to the functionality provided by the functions and the problems they solve. To assess functionality, we will first look at the function definition, the encapsulating struct or trait and the documentation. If after going through these, the exact functionality is still not clear, or if the documentation is missing, we will look at the source code. In this process, we will carefully try to balance accuracy and correctness while staying efficient and not spending too much time on singular functions.

We will keep track of this tree structure and our defined rulesets in a work file. This file will also be published to the recreation package such that other people can easily follow it and perform the same categorization we did, without the need for redefining a categorization tree.

Throughout the analysis process, we write our findings and categorization tree to the DuckDB file to combine these with our earlier static analysis results. At the end, we also write our categorization tree to this file such that we may use it when writing our analysis SQL queries.

3.6 Analysis of DuckDB Findings

After the necessary data has been gathered through our static and functional analysis, we will be left with a DuckDB file. This file will contain all of the necessary information of the underlying source code we analyzed like the categorizations of the functions and how these relate to one another.

To formulate the answers to our research questions, we will then create a DuckDB notebook to analyze these results. This will contain a set of cells for each research question, in such a way that each cell is responsible for one number or statistic in our report. We will include a brief explanation above each cell explaining what metric this query is responsible for.

This notebook can be found in our recreation package and makes it very easy for other people to test and verify the results we use in our report. Furthermore, this should also make it easier to expand our analyzed dataset, since these queries are quick and easy to rerun at any time.

4 Rustc-Analysis Tool

This section explains how we implemented our Rustc-Analysis tool. The source code and accompanying README can be found in our public recreation package [5].

4.1 Problem Statement and Requirements

Problem

In this research paper, we try to formulate generalizable answers to our research question. In a best-case scenario, these findings should be relevant to all publicly available Rust code. As of today, if we limit ourselves to GitHub, that would imply 1,430,000 repositories [20]. This is clearly an infeasible amount to analyze, but it does raise the question: how can we maximize the code analyzed?

While a manual analysis might be more straightforward, it introduces some problems. It would make reproducing our findings significantly more difficult as well and limiting the amount of data we could analyze. Our automated approach solves both of these issues.

Input Data

The input to our analysis tool should be a list consisting of the GitHub URLs and hashes to the repositories and commits we analyzed. Furthermore, if compiling the repository requires specific Cargo arguments, these should also be included. This ensures consistent replication, allowing people to share their lists to rerun the analysis and reproduce reported findings.

Output Data

Before implementing our analysis tool, we need to consider what output data is required to answer our research questions. To answer RQ1, we chose the crate's name and GitHub repository link. This URL adds info about the crate owner, for example `tokio-rs`, `smol-rs` or `async-rs`. Furthermore, to capture the structure of the code, our output data should include a dependency/call graph. This allows us to write queries to analyze asynchronous where and how asynchronous code is being used.

4.2 Architecture

Source Code Representation

The source code representation used would play a crucial role in the development and validity of our tool. In section 2.4,

we provided an explanation regarding the available representations. In our development process, we considered all layers starting at the raw, textual input. However, the lack of semantic understanding at these levels quickly had us descending deeper into the Rust compiler internals.

The following is a brief summarization of tools we considered and their respective representations: textual source code, AST using Syn, HIR/MIR using Rust-Analyzer, MIR using stable-mir/rustc-public, MIR using Charon [14] and LLVM IR using LLVM. While all of these seemed promising, none of them fit our requirements for semantic understanding and large-scale analysis. Specifically Charon piqued our interest since it is aimed at analyzing Rust crates using simplified MIR bodies while abstracting away compiler internals. Unfortunately, Charon does not support async functions, and explicitly mentions this as an out-of-scope feature [8].

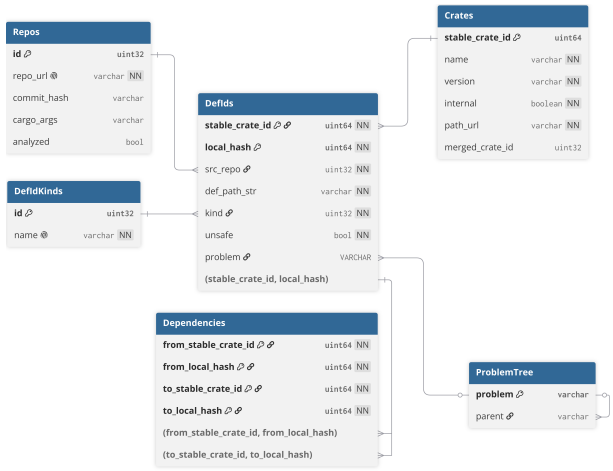


Figure 3: Entity Relationship Diagram of Dataset

Output Data Representation

Given our output data requirements, we had to decide how we might represent them. This is where we considered a comma-separated-value representation vs. a relational-database representation. Using the latter would allow integration with SQL frameworks, providing excellent performance and providing advanced query capabilities. Furthermore, our data was very relational in essence, with entities like Crates, Repositories, DefIds and Dependencies. Given these considerations, we opted for the relational approach and came up with the entity-relationship diagram shown in figure 3.

Finally, we had to choose a framework for our database where we considered SQLite, DuckDB and PostgreSQL. We were looking for a simple framework, with good analytical performance and capabilities. This lead us to DuckDB which is aimed at providing easy and fast analytical capabilities [6].

4.3 Design and Implementation

Sequence Diagram

Our solution implements a control flow graph shown in figure 4. Control starts from the user, who invokes the user-facing binary. This is where the command-line arguments are

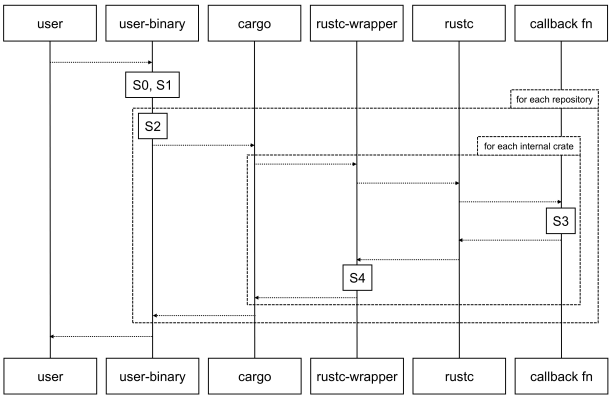


Figure 4: Application Control Flow Graph

parsed and where the required pipeline stages are run. For the initialization, this is stage 0. For the analysis, this involves stage 1 and 2. In the latter case, control is handed over through an invocation to Cargo including our rustc wrapper. Cargo invokes this at which point necessary metadata is loaded and the compiler is invoked using our callback function. This is where the analysis starts (stage 3). After this, control is handed back to the rustc-wrapper and the results are written to the DuckDB output file (stage 4). Finally, control is returned to Cargo and this process is repeated for each of the internal crates. Finally control is handed back to the user binary which will repeat this process for each of the repositories. After all of these have completed, the analysis is done and control is handed back to the user.

S0 : Setup and Initialization

These are the first stages in our tool. The first stage is responsible for creating and initializing the DuckDB file using our database scheme. The second then fetches a number of URLs of most-starred Rust repositories as specified by the user and writes these to the DuckDB file. These stages are optional and may be skipped in case a list of repositories is being reused

S1: Fetching Source Code

From the generated/shared DuckDB file, we read URLs and hashes defining a unique repository commit. We then fetch said source code and save it to a folder specified by the user.

S2: Gathering Necessary Metadata

An issue we encountered with rustc was a lack of crate metadata available to us. In the rustc environment, we only have access to the compile-target name and the file location of binaries. Since we need more info for our analysis like the crate name, version and whether it is internal/external, we need Cargo metadata data.

This was solved by gathering and passing this data through the filesystem to the callback function on the rustc side. Stage 2 is responsible for gathering and persisting this metadata to the filesystem. On the rustc side, we have implemented logic which matches binaries back to their source crate.

S3: Analysis using Compiler

Our callback function analyzes the functions and closures defined in the HIR. We fetch their MIR bodies and extract the

calls they made. These are then used to construct a call graph.

Finally, we gather necessary auxiliary data from rustc and construct a Rust version of the database. Since the rustc representations are fairly rough, this requires extracting and matching logic to get the expected data format.

S4: Writing our Output Data

The Rust database structure matches our schema, allowing us to easily write it to the DuckDB output file. If this was successful for all internal crates, the repository is marked as analyzed and will not be analyzed again in future invocations.

4.4 User Flow and User Guide

To use our tool, the user must have git and a rust development toolkit installed. The tool provides two commands: `init-repo-list` and `analyze`. These commands reflect the order in which the analysis is usually run.

Initialization

The `init-repo-list` command is optionally run first. It creates the DuckDB file and initializes it with the correct database schema. Furthermore, `init-repo-list` allows the user to specify the number of repositories they wish to analyze. The tool will then fetch the metadata of the top-n repositories from GitHub and write these to the DuckDB file.

It is important to note that the `init-repo-list` command is optional. To make the results reproducible, people may share this intermediate DuckDB file. At this point, it includes a list of repository URLs, the used commit hash and any Cargo arguments. Anyone interested can use this data to fetch the same source code, and run the compiler with the same arguments, to obtain identical behavior and output.

Verification

Before running the analysis, it is important that users verify repositories compile successfully. This is necessary because we depend on the Rust compiler's MIR and HIR presentation. If the Rust compiler is unable to compile the source code, it cannot construct these, resulting in an error from rustc.

This can be verified by running `Cargo check` on each repository and ensuring this passes. A script and explanation on how to do this can be found in the recreation package.

Analysis

The `analyze` command is responsible for performing the actual analysis. It gets the repositories and corresponding Cargo arguments from the DuckDB file and invokes the callback on these individually. Our tool then analyzes the internal crates defined in these repositories. Finally, it writes back the results to the DuckDB file and marks the repository as analyzed. If the analysis fails on any of the repositories, the tool will not mark this repository as analyzed, and subsequent invocations will only analyze unanalyzed repositories.

5 Results and Discussion

5.1 Dataset Overview

The dataset we used was made from the top-25 most popular rust repositories on GitHub, of these, 2 were excluded due to them not being applications and 7 were excluded since they failed to build. This left us with a total of 16 repositories.

Within these repositories, we discovered and analyzed 510 unique **internal** crates. The repositories used 2,593 unique **versions** of **external** crates. After merging these along different versions, we ended up with a total of 1,649 **unique external** crates used. Crates with the most distinct versions were hashbrown and winnow with 9, and toml, getrandom and bitflags with 8 distinct versions.

In total, our dataset found 9,106 function and closure definitions involved in asynchronous handling. Of these, 275 asynchronous functions were defined in external crates and called by our codebase. Of the remaining 8,831 internally defined functions, 6,124 were asynchronous while 2,707 were synchronous functions that simply interacted with synchronous functions.

We analyzed 13,711 async invocations. These consisted of 11,270 invocations to targets defined in internal crates and 2,441 invocations to targets defined in external crates.

5.2 RQ1: What are the most popular runtimes?

Runtime	Direct	Indirect	crates.io
tokio	7	11	60,275
futures	7	9	22,429
async-std	0	0	1,936
smol	0	0	636

Table 1: Runtime usage statistics

Data Explanation

The table above demonstrates runtime usages based on three distinct metrics: direct dependencies, indirect dependencies and `crates.io` dependencies.

Direct dependencies refer to the number of repositories that explicitly used said runtime in their source code with a `use` statement. Out of our 16 repositories, 2 repositories used only tokio, 2 repositories used only futures, and 5 repositories used both tokio and futures in their source code.

Indirect dependencies refer to repositories where one of these runtimes was used directly, or indirectly through one of its dependencies. These results again show a clear prominence in tokio and futures usage.

`Crates.io` dependencies refer to the dependencies as reported by `crates.io` using reverse dependency lookup.

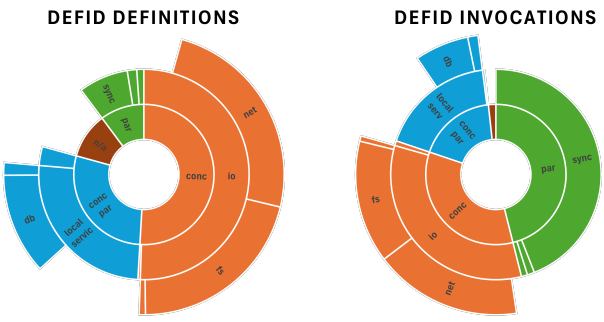
Conclusion

All three of our dependency types show a clear agreement in popularity. Tokio is the most dominant runtime used in Rust, with futures also taking a prominent role while async-std has a significantly lower usage. This makes sense, as it has been discontinued as of March 1st, 2025. Smol is the least used runtime and did not show up in any of our analyzed repositories. This might be because it provides a smaller, more minimalistic implementation, fit for niche applications.

5.3 RQ2: What functionality do external async APIs provide?

Data Explanation

The illustrations above show a distribution of external functions and APIs used by our codebase.



(a) API definition (b) API invocation

Figure 5: Comparison of API definition and invocation distributions.

Figure 5a shows a distribution of our categorization results regarding distinct asynchronous external API endpoints used by our codebase. Figure 5b uses the same categorization, but rather looks at how significantly these API endpoints were used and invoked.

From both diagrams, it becomes clear that a large fraction of these asynchronous external API endpoints are used to prevent blocking issues. We defined these in our categorization as "concurrent". These are mainly related to file-system operations (reading/writing) as well as handling and implementing network connections (HTTP/TLS/DNS). Networking and file-system functions make up respectively 24.3% and 21.1% of API definitions, and make up respectively 17.0% and 14.1% of API invocations.

Synchronization makes for another large fraction of these external API uses. Interestingly enough, synchronization only makes up 7.6% of distinct API functions, yet is responsible for 44.0% of all external API calls. This shows its importance, regardless of its small exposure surface. Logically, this makes sense as this categorization is responsible for synchronization between threads and clearly from our data, this needs to happen a lot. It does this synchronization using structures like locks, mutexes, barriers and more.

Concurrent parallel is our final categorization. It was aimed at endpoints which are more high level and require/provide both concurrency and parallelism. This often involved local servers or applications providing a service, hence our local service categorization. Examples that fell under this categorization were local database servers, libraries providing a local language server and testing engines. As we might have speculated, these offer a wide array of endpoints, seeing as these are very application and library specific. In the end, local service endpoints provided 25.5% of distinct endpoints, while being called in 17.8% of API invocations.

Conclusion

The most commonly used external asynchronous APIs are responsible for synchronization between threads and input/output operations to the local file system and over the connected network. A smaller part of asynchronous APIs provide a local service like database engines or language servers.

6 Responsible Research

We tried to create our methodology with ethical values and responsible research in mind. In this section, we outline some of our considerations and how we addressed these.

6.1 Reproducibility

We created our methodology with reproducibility as a core value. We have published a recreation package on GitHub [5]. This recreation package includes a README with further explanation on how we performed our analysis. It lays out a step-by-step guide on how others can create the same results and findings as we did and draw similar conclusions. The README also explains the architecture of the tool we made and used to generate the DuckDB output file. It explains the different commands offered and the pipeline this activates. Finally, the recreation package also includes a DuckDB file with a list of repositories and commit-hashes used in our dataset. Using the analysis tool on this DuckDB file will automatically fetch the same code we used and generate the same DuckDB file we used to answer the questions in this paper.

6.2 Threats to Validity

Correctness Analysis Tool

Since we developed a custom analysis tool, there is a risk to correctness regarding its implementation. To minimize this, we wrote a small test crate using basic asynchronous functions and closures and manually verified the tool's results matched our expectations. We also made it interact with other crates and tried introducing edge cases like cyclical dependencies to see how the tool would handle these. We then also tested the tool on some GitHub repositories, and verified its results. We did this by comparing the resulting call graph, function data and crate data to the original source code and ensuring all of these matched. This provided us with the necessary confidence in the tool to continue using it in our analysis. As a future improvement, an automated test suite would be preferred, but we feel that this falls outside the scope of this 10-week research project.

Data Subset and Generalization

In this paper, we analyzed a subset of open-source repositories which might not be generalizable to all Rust projects. As an example, we might be missing out on smaller, more obscure applications like microcontrollers and custom runtimes. To analyze these, a randomly sampled dataset would be preferred. We deliberately chose not to do this, since we wanted to be able to reason about code with the highest impact on end-users, which meant looking at popular Rust crates. However, this could still reveal interesting findings and thus, we have added it as a future suggestion in subsection 8.3.

Bias

While analyzing the data, we excluded repositories which we could not compile successfully, since this is required by rustc. We assessed this by running `cargo check` and seeing if it exited successfully. In repositories where this was not the case, we could often resolve this without much hassle, for example by making sure necessary dependencies were installed. However, some repositories showed complex errors in their or

their dependencies' source code. In these cases, we checked the README file and searched online. If we were unable to resolve the issue within 15 minutes, we considered the repository failing/unstable and excluded it from our dataset. These issues arose at random and did not appear correlated to project size, complexity or field. Hence, we do not consider this to skew our dataset in a meaningful way.

6.3 Ethical Aspects

Data Source and Redistribution

For our analysis, we only used publicly available, open-source code and ensured not to use or accidentally redistribute any private information or data. Furthermore, in our recreation package, we do not store any code or datasets. We instead store repository URLs and commit hashes. By doing so, the code authors can always remove or privatize their code without us accidentally leaking their privatized repository.

7 Related Work

7.1 Safety in Rust

Multiple empirical studies have analyzed bugs and vulnerabilities caused by concurrent code. Zhen et al. [41] looked at security risks in the Rust ecosystem and found that concurrency was the second biggest cause of vulnerabilities after memory issues, and was responsible for 20.56% of all vulnerability bugs. Qin et al. [17] specifically analyzed 100 concurrency bugs to uncover common pitfalls in asynchronous Rust code. They found the most common causes to be deadlocks and data races and report that Rust's compiler checks do not adequately cover all types of bugs.

Cook et al. [4] present the largest verification campaign for a software library to date, analyzing a subset of the standard library using verification tools. Since the standard library relies heavily on unsafe code, they try to formally prove the absence of undefined behavior. However, they report concurrency still remains as a significant challenge, and still requiring future work regarding atomic types, `Arc`, and the standard libraries lock-free types.

Further research into deadlocks is conducted by Zhang et al. [40]. They continue on the work of Qin et al. [17] and Yu et al. [39] by providing RcChecker, a static detection method to identify resource and communication deadlocks, eliminating false positives and discovering new errors, previously missed by existing tools.

These works show that safety concerns regarding concurrent Rust code still exist, especially involving deadlocks and data races. They highlight the importance of continued research into the asynchronous ecosystem and show there are still many things to be learned and improved upon.

7.2 Performance Analysis

Ivanov [9] conducts a comparative performance benchmark regarding commonly used algorithms in Rust and C++ and reports their performance to be similar, with Rust being slightly faster in some, and slightly slower in others.

Besides this, some community-driven benchmarks have taken place. Karneges [11] analyzed the runtime overhead caused by async Rust in an attempt to verify Rust's "zero-cost abstraction" claim [26], reporting there to be a cost to the encapsulation, but also stating this to be negligible in actual production code.

Developers of the Eclipse Zenoh project [23] compared performance of tokio, async-std and smol using ping-pong round-trip time (RTT). They found async-std and smol to achieve lower RTT than tokio under CPU-bound asynchronous tasks. However, the evaluation was limited to a specific workload, not peer-reviewed and has become somewhat outdated, as async-std has been discontinued and projects have been recommended to use smol instead.

While these studies have investigated the performance characteristics of Rust, comparatively little academic work has focused on the performance of asynchronous Rust and the overhead of different runtimes. This suggests opportunities for further research, particularly into the trade-offs between different asynchronous runtimes.

7.3 Static Analysis Tools

In the past few years, multiple tools aimed at analyzing Rust code have been developed and published. One of these tools we were particularly interested in was Charon by Ho et al. [8]. They proposed a framework aimed at analyzing Rust programs, providing an AST as a foundation for further research. Unfortunately, Charon does not support async, and explicitly mentions this as an out-of-scope feature.

Jung et al. [10] published Miri, a Rust MIR interpreter, aimed at discovering all de-facto undefined behavior in deterministic Rust programs. It's primarily a correctness and debugging tool, aimed at catching data races and discovering bugs in unsafe code.

Many more analysis tools have been proposed like RustMC by Pearce et al. [15], hax by Bhargavan et al. [3], Crux by Pernsteiner et al. [16], Verus by Lattuada et al. [12] and Rudra by Bae et al. [2]. These tools and verifiers primarily detect bugs and attempt to prove the correctness of underlying code.

While these solutions are capable of detecting bugs and verifying correctness, they are not aimed at extracting the information we require to answer our research questions.

7.4 Rust Ecosystem Studies

Other empirical Rust studies mainly looked at how and why unsafe code was used in the Rust ecosystem [1; 7; 18]. These studies are over six years old, which raises concerns regarding their relevancy to the modern Rust ecosystem.

Out of these three, we want to briefly point out the analysis methodology used by Qin et al. [18]. They proposed Qrates, which analyzed code through the Rust compiler using a callback function, which they refer to as a "plugin". This plugin is used to construct a database, using the CFG representation provided by the MIR. They then answer their research questions by running queries on this database. This is similar to the approach used in this paper. Unfortunately, Qrates heavily depends on internal rustc APIs which are explicitly not covered by stability guarantees [31]. Due to the size and age of this tool, we considered the effort required to validate and adapt it to modern Rust to be outside the scope of this study, making it unfit to be used in our analysis.

8 Future Work and Recommendations

8.1 Use of our Tool and Findings

Alongside this research paper, we provide a rustc-based analysis tool. The tool is able to analyze Rust code and extract information like call graphs, function information, crate information and more. We encourage others to try out this tool and give full permission to reuse or improve upon this tool.

Furthermore, our findings highlight the most common fields where external asynchronous APIs are used. This might be helpful to Rust developers interested in learning asynchronous programming. It might also encourage them to try out tokio, seeing as it is the primary runtime used by the repositories we analyzed.

Furthermore, it highlights the significance of synchronization and IO operations. This might guide future research into analyzing these specifically as they are the most prevalent. We emphasize the importance of assuring these two fields function properly, as a lot of repositories will depend on them.

8.2 Cargo-Rustc Crate Referencing

While developing our tool, a significant challenge we faced was the identifying and linking of crates defined in rustc to their original source crate.

In our analysis, we required metadata about crates referenced by rustc. This included their name, version and repository URL. None of these are available in the rustc presentation. This meant we had to manually match internal rustc crates with their public cargo metadata presentations.

As an improvement to the Rust toolchain, we would like to propose a stable, unique identifier, like rustc's `stable_crate_id`, to be made available publicly through cargo metadata. This would improve accessibility for future tools, which require more information about crates than currently available through rustc alone.

8.3 Expand Input Data

A limitation to our research was the scale of the used dataset. We start from the 25 most popular Rust repositories, 16 of which met inclusion criteria. Out of these 16, 8 used asynchronous constructs. For more reliable and generalizable claims to be made, a larger dataset would be preferred.

Therefore, we would like to suggest continuing our analysis as explained in section 3 on a larger dataset of repositories using asynchronous Rust.

While analyzing more repositories, one might also consider employing a different selection algorithm. In this report, we prioritized the most impactful repositories by ranking repositories using stars. In conducting further research, one might consider switching to a random sampling algorithm instead, as to obtain a more uniform dataset.

The majority of time required for continuing this research would be spent on manually ensuring all new repositories compiled successfully and subsequently performing the manual analysis as explained in section 3.5. However, most means to perform this have already been established and may be reused freely, like the analysis tool and categorization ruleset. These should make future work significantly easier and remove some of the risks we faced.

8.4 Internal Categorization Using LLMs

Through our current methodology, we are severely limited by the manual categorization. It is very labor intensive and requires reading each function's documentation and/or source code. Our dataset alone contains 9,106 function definitions, making it infeasible to categorize all of them.

Luckily, 8,831 of these were defined internally meaning we have access to their source code. Furthermore, rustc can provide the exact line numbers of function definitions through the `def_span` API. This provides us with a unique opportunity to automatically extract and parse this source code through an ML model to automatically categorize them.

This raises a lot of questions regarding accuracy and reproducibility. Furthermore, we also have to consider what kind of model we would use. Would we simply use an existing LLM or train our own categorization model? What kind of data could we use to train this model?

While we believe this to be a useful and interesting addition to our analysis tool, we also understand its scale and complexity. Due to the time constraints of our research project, we have decided to suggest it as a future improvement instead.

9 Conclusion

This study presented Rustc-Analysis, an automated analysis tool for discovering and extracting structural information from asynchronous Rust code. We looked at the functionality provided by external asynchronous APIs and found them to be primarily used for synchronization between concurrent tasks and I/O operations, namely networking and file-system interactions. We analyzed runtime adoption, and found tokio to be the most commonly used runtime, followed by futures, async-std and smol. These findings provide empirical insight into how asynchronous Rust is used in practice. However, the dataset used in this analysis was relatively small and future work should aim at expanding the input data, as well as broadening our research to analyze the functionality provided by internal asynchronous functions.

References

[1] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. How do programmers use unsafe rust? *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–27, November 2020. doi:10.1145/3428204.

[2] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. Rudra: Finding memory safety bugs in rust at the ecosystem scale. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 84–99. ACM, October 2021. doi:10.1145/3477132.3483570.

[3] Karthikeyan Bhargavan, Maxime Buyse, Lucas Franceschino, Lasse Letager Hansen, Franziskus Kiefer, Jonas Schneider-Bensch, and Bas Spitters. hax: Verifying security-critical rust software using multiple provers. Cryptology ePrint Archive, Paper 2025/142, 2025. URL: https://eprint.iacr.org/2025/142.

[4] Byron Cook, Remi Delmas, Zyad Hassan, Bart Jacobs, Ranjit Jhala, Rahul Kumar, Felipe R. Monteiro, Thanh Nguyen, Rebecca Rumbul, Michael Tautschnig, Celina Val, and Carolyn Zech. Verifying the rust standard library. In Jyotirmoy Deshmukh, Klaus Havelund, and Alessandro Pinto, editors, *NASA Formal Methods*, pages 415–435, Cham, 2026. Springer Nature Switzerland.

[5] Benjamin Dockx. rustc-analysis: Recreation package, 2026. URL: https://github.com/Benjamin1260/rustc-analysis.

[6] DuckDB Foundation. Why duckdb, 2026. URL: https://duckdb.org/why_duckdb.

[7] Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. Is rust used safely by software developers? In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 246–257, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3377811.3380413.

[8] Son Ho, Guillaume Boisseau, Lucas Franceschino, Yoann Prak, Aymeric Fromherz, and Jonathan Protzenko. Charon: An analysis framework for rust, 2025. URL: https://arxiv.org/abs/2410.18042, arXiv:2410.18042.

[9] Nikolay Ivanov. Is rust c++-fast? benchmarking system languages on everyday routines, 2022. URL: https://arxiv.org/abs/2209.09127, arXiv:2209.09127.

[10] Ralf Jung, Benjamin Kimock, Christian Poveda, Eduardo Sánchez Muñoz, Oli Scherer, and Qian Wang. Miri: Practical undefined behavior detection for rust. *Proceedings of the ACM on Programming Languages*, 10(POPL):1383–1411, January 2026. doi:10.1145/3776690.

[11] Justin Karneges. Rust async benchmark, 2022. URL: https://github.com/jkarneges/rust-async-bench.

[12] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying rust programs using linear ghost types. *Proc. ACM Program. Lang.*, 7(OOPSLA1), April 2023. doi:10.1145/3586037.

[13] Niko Matsakis. Async-await on stable rust!, 2019. URL: https://blog.rust-lang.org/2019/11/07/Async-await-stable/.

[14] Nadrieril, Sonmarcho, n1ark, and Sam-Ni. Charon, 2021. URL: https://github.com/AeneasVerif/charon.

[15] Oliver Pearce, Julien Lange, and Dan O'Keeffe. Rustmc : Automated verification of real-world concurrent rust. In Laura Bocchi and Burcu Kulahcioglu Ozkan, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, pages 235–255, Cham, 2026. Springer Nature Switzerland.

[16] Stuart Pernsteiner, Iavor S. Diatchki, Robert Dockins, Mike Dodds, Joe Hendrix, Tristan Ravich, Patrick Redmond, Ryan Scott, and Aaron Tomb. Crux, a precise verifier for rust and other languages, 2024. doi:10.48550/ARXIV.2410.18280.

[17] Boqin Qin, Yilun Chen, Haopeng Liu, Hua Zhang, Qiaoyan Wen, Linhai Song, and Yiying Zhang. Understanding and detecting real-world safety issues in rust. *IEEE Transactions on Software Engineering*, 50(6):1306–1324, 2024. doi:10.1109/TSE.2024.3380393.

[18] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. Understanding memory and thread safety practices and issues in real-world rust programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 763–779, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3385412.3386036.

[19] Runtime Verification. Enhancing stable mir with serde serialisation, 2024. URL: https://runtimeverification.com/blog/enhancing-stable-mir-with-serde-serialisation.

[20] Rust Project Developers. rust-repos, 2026. URL: https://github.com/rust-lang/rust-repos/tree/master.

[21] Tom Schroeder, Minh Phan, and Yang Chen. A preliminary study of fixed flaky tests in rust projects on github, 2025. URL: https://arxiv.org/abs/2502.02760, arXiv:2502.02760.

[22] Stack Overflow. 2025 developer survey: Technology, 2025. URL: https://survey.stackoverflow.co/2025/technology.

[23] the Eclipse Foundation. A performance evaluation on rust asynchronous frameworks, 2022. URL: https://zenoh.io/blog/2022-04-14-rust-async-eval/.

[24] The Rust Project Developers. Feature: Async Await, 2018. URL: https://rust-lang.github.io/rfcs/2394-async_await.html.

[25] The Rust Project Developers. Task Wakeups With Waker, 2021. URL: https://rust-lang.github.io/async-book/02_execution/03_wakeups.html.

[26] The Rust Project Developers. Why Async?, 2021. URL: https://rust-lang.github.io/async-book/01_getting_started/02_why_async.html.

[27] The Rust Project Developers. The MIR (Mid-level IR), 2025. URL: https://rustc-dev-guide.rust-lang.org/mir/index.html.

[28] The Rust Project Developers. Async functions, 2026. URL: https://doc.rust-lang.org/reference/items/functions.html#async-functions.

[29] The Rust Project Developers. Await, 2026. URL: https://rust-lang.github.io/async-book/part-guide/async-await.html#await.

[30] The Rust Project Developers. Overview of the compiler, 2026. URL: https://rustc-dev-guide.rust-lang.org/overview.html.

[31] The Rust Project Developers. Stability guarantees, 2026. URL: https://rustc-dev-guide.rust-lang.org/stability-guarantees.html.

[32] The Rust Project Developers. Syntax and the ast, 2026. URL: https://rustc-dev-guide.rust-lang.org/syntax-intro.html.

[33] The Rust Project Developers. The HIR, 2026. URL: https://rustc-dev-guide.rust-lang.org/hir.html.

[34] The Rust Project Developers. Trait future, 2026. URL: https://doc.rust-lang.org/core/future/trait.Future.html.

[35] The Rust Project Developers. Trait future, 2026. URL: https://docs.rs/futures/latest/futures/future/trait.Future.html.

[36] The Rust Project Developers. What is a control-flow graph?, 2026. URL: https://rustc-dev-guide.rust-lang.org/appendix/background.html#cfg.

[37] Wikipedia contributors. Semantics (programming languages), 2026. URL: https://en.wikipedia.org/w/index.php?title=Semantics_(programming_languages)&oldid=1349125305.

[38] Wikipedia contributors. Syntax (programming languages), 2026. URL: https://en.wikipedia.org/w/index.php?title=Syntax_(programming_languages)&oldid=1338087743.

[39] Zeming Yu, Linhai Song, and Yiying Zhang. Fearless concurrency? understanding concurrent programming safety in real-world rust software. *CoRR*, abs/1902.01906, 2019. URL: http://arxiv.org/abs/1902.01906, `arXiv:1902.01906`.

[40] Yu Zhang, Kaiwen Zhang, Guanjun Liu, Yuandao Cai, and Shengchao Qin. Two birds one stone: Effective static detection of resource and communication deadlocks in rust programs. *Automated Software Engineering*, 33(2), March 2026. `doi:10.1007/s10515-026-00614-z`.

[41] Xiaoye Zheng, Zhiyuan Wan, Yun Zhang, Rui Chang, and David Lo. A closer look at the security risks in the rust ecosystem. *ACM Transactions on Software Engineering and Methodology*, 33(2):1–30, December 2023. `doi:10.1145/3624738`.

A Repository Host Distribution

Table 2 shows the distribution of hosting services used for the top-10,000 crates on `crates.io`. These results were generated 19 May 2026.

Repository Host	Count
GitHub	9597
Gitlab	145
Codeberg	45
No repository listed	179
Other	34

Table 2: Hosting distribution of the top-10,000 crates.

B GitHub Repository Presence on `crates.io`

Table 3 shows the amount of the top-x GitHub projects which could also be found in the top-10,000 crates on `crates.io`. These results were obtained on 19 May 2026.

Top-x GitHub	On `crates.io`	percentage
25	10	40.0%
250	68	27.2%
1000	254	25.4%

Table 3: Presence of GitHub repositories on `crates.io`.